# Multibit Flip-Flop Banking via Compact Placement and Iterative Matching

Yewei Wang
HKUST Guangzhou

Fangyao Chen
HUST

Shiju Lin*
HKUST Guangzhou

## ABSTRACT

Power consumption has become a critical concern in advanced technology nodes due to its significant impact on thermal management and the reliability of integrated circuits. An effective strategy for reducing power consumption is to bank (i.e., merge) multiple single-bit flip-flops (SBFFs) into a multi-bit flip-flop (MBFF), which achieves lower per-bit power consumption. However, existing MBFF banking algorithms typically cluster SBFFs into MBFFs and then relocate them, leading to fragmentation of available space and underutilization of spatial resources. In this work, we propose a novel approach that first compactly places as many MBFFs as possible on the circuit, and then maps the original SBFFs to these MBFFs. This paradigm shift enhances space utilization, allowing for a greater number of SBFFs to be banked into MBFFs, thus further reducing power consumption. Experimental results show that our method outperforms state-of-the-art techniques on the ICCAD Contest 2024 benchmarks. The source code of this work is available at https://github.com/Lin-HKUST-Guangzhou/mbff-banking.

## 1 INTRODUCTION

As technology nodes continue to shrink, the number of transistors packed into a single chip increases significantly, which in turn raises the power consumption of the chip. This escalating power density not only heightens concerns about thermal management but also affects the reliability of integrated circuits. Consequently, power optimization has become a critical focus for integrated circuits, particularly at advanced technology nodes.

A substantial portion of a chip's power consumption is attributed to the clock network. One effective strategy for power reduction is to bank (i.e., merge) multiple single-bit flip-flops (SBFFs) into a multi-bit flip-flop (MBFF) [3–5, 11–13]. Figure 1a illustrates an example where two SBFFs are banked into a 2-bit FF. The advantages of using MBFFs are twofold. Firstly, MBFFs offer reduced per-bit area and power consumption compared to SBFFs. As shown in Figure 1a, a 2-bit FF requires one fewer clock inverter than two SBFFs, resulting in area and power savings. Secondly, banking SBFFs into an MBFF simplifies the clock tree topology, thereby reducing the power consumption of the clock tree, as demonstrated in Figure 1b. However, despite the potential to reduce power and area, MBFF banking can adversely impact timing due to the relocation of SBFFs during the process, especially when the SBFFs being banked are far apart. Therefore, an effective banking solution should consider multiple objectives, such as timing, area, and power, to achieve the optimal tradeoff.

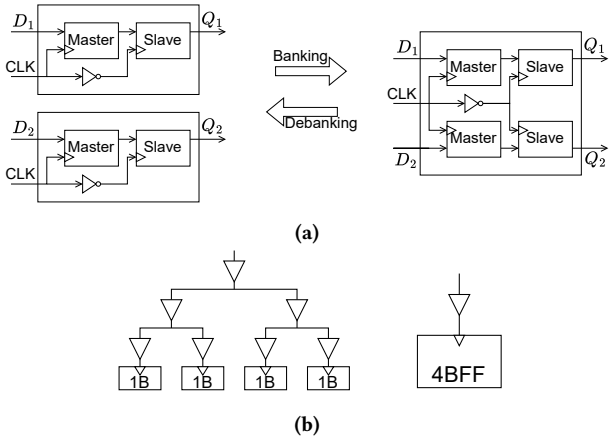
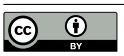



**Figure 1: (a) MBFF banking example: banking two single-bit FFs to a 2-bit FF. (b) Clock network comparison between 4 single-bit FFs and a 4-bit FF.**

State-of-the-art MBFF banking methods typically follow a similar paradigm involving *SBFF clustering* followed by *MBFF relocation*. SBFF clustering identifies closely located SBFFs to form a cluster for banking, while MBFF relocation determines a suitable location for the newly banked MBFF. For example, Wu et al. [13] proposed a weighted K-means algorithm for SBFF clustering, and Hsieh et al. [3] introduced an effective mean shift technique to naturally form SBFF clusters. However, this paradigm suffers from poor space utilization, primarily due to the lack of cooperation between the relocation of different MBFFs. Each SBFF cluster, representing an MBFF, seeks the best location for itself without considering the impact of occupying that location on other MBFFs. Subsequently, the empty space is highly fragmented and cannot be used for placing MBFFs. To improve the space utilization and allow more SBFFs to be banked, we propose a new banking method with the following key contributions:

- We present a new framework that first compactly places MBFFs, followed by an iterative process for matching each SBFF to a nearby MBFF. This approach prioritizes MBFF placement over previous methods, enabling the accommodation of more MBFFs and increasing opportunities for power reduction.
- We propose a robust method for matching original SBFFs with newly created MBFFs. This method takes into account clock-net constraints, total negative slack (TNS) minimization, matching quality, and banking efficiency, all through an iterative negotiation process.

*Corresponding author (shijulin@hkust-gz.edu.cn)



This work is licensed under a Creative Commons Attribution 4.0 International License.
*ICCAD '26, November 08–12, 2026, San Jose, CA, USA*
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2873-0/2026/11
https://doi.org/10.1145/3831252.3834140

- We introduce refinement strategies based on displacement and TNS that effectively address excessive TNS increases during MBFF banking.
- We conducted extensive experiments to validate our proposed method. The results demonstrate improvements of 2% and 20% compared to the first-place team in the 2024 ICCAD Contest [14] and TIMBER [1]. This work will be made open-source.

The remainder of this paper is organized as follows. First, Section 2 introduces the preliminaries of MBFF banking and outlines the problem description. Next, Section 3 details our proposed method. Finally, Section 4 presents the experimental results, followed by the conclusions in Section 5.

## 2 PRELIMINARIES

### 2.1 Flip-Flop Banking and Debanking

A flip-flop (FF) is a fundamental storage element in sequential circuits, storing binary data and updating state via control signals. A multi-bit FF (MBFF) is an FF variant that integrates multiple storage bits (e.g., 2–4 bits) in a single structure, sharing clock/reset circuitry to reduce area and power consumption.

FF banking refers to replacing multiple single-bit FFs (SBFFs) with an MBFF to save area and reduce routing complexity of power, ground, and clock nets. FF debanking refers to splitting a MBFF into several SBFFs to optimize timing-critical nets, addressing potential timing degradation caused by banking.

### 2.2 Problem Description

FF banking has different constraints and applications across the electronic design automation (EDA) flow. This paper specifically focuses on the problem formulation introduced in Problem B [14] of the 2024 CAD Contest at ICCAD [2].

**Problem Statement.** Given a placed design and a cell library consisting of various FFs, we aim to improve the design by relocating, banking and debanking FFs. Note that only FFs are allowed to move and combinational gates should remain in place. Solutions are evaluated by a weighted sum of total negative slack (TNS), power consumption, area and violations of bin utilization, with details explained below.

**Power and Area Calculation.** The area and power of each FF in the cell library are characterized, with typical values provided in the problem input. The total area and power are calculated as the sums of the individual values.

**TNS Calculation.** The total negative slack (TNS) is determined by summing the negative slacks of all D-pins of the FFs. Instead of calculating the slack of an FF D-pin directly, it is estimated based on the initial slack value provided in the input and the change in half-perimeter wirelength (HPWL) of its critical path. Specifically, for an FF D-pin $d$, we consider all timing paths that start from an FF Q-pin and end at $d$. Figure 2 illustrates this with two timing paths. The critical HPWL for $d$ is defined as the maximum HPWL among all these timing paths, where the HPWL of a single path is calculated as the sum of the half-perimeter wirelengths for each pair of connections. As shown in Figure 2, path 1 is identified as the critical path, with three pairs of connections, $S_1$, $S_2$ and $S_4$. The

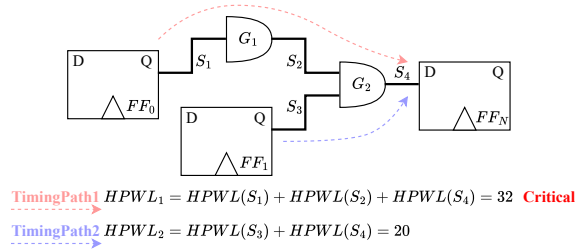



**Figure 2: HPWL calculation**

new slack is then calculated by[1]

$$\texttt{newSlack} = \texttt{initSlack} + \Delta\texttt{criticalHpwl} \times \texttt{displacementDelay}$$

where `initSlack` is the initial slack value of $d$ provided in the input, $\Delta$`criticalHpwl` is the difference between the current and initial critical HPWL, and `displacementDelay` is a coefficient for slack estimation given in the input.

**Bin Utilization.** The die is divided into several areas called bins. The utilization of a bin refers to the percentage of the area occupied by both FFs and combinational cells within the bin. A maximum utilization is provided for each bin and violating this constraint will result in certain penalty.

**Clock Net Constraint.** Only SBFFs from the same clock net can be banked together. This is to ensure the correctness of the design after banking.

## 3 METHOD

In this section, we first present an overview of our method in Section 3.1, followed by a detailed discussion of its individual components in the subsequent subsections.

### 3.1 Overview

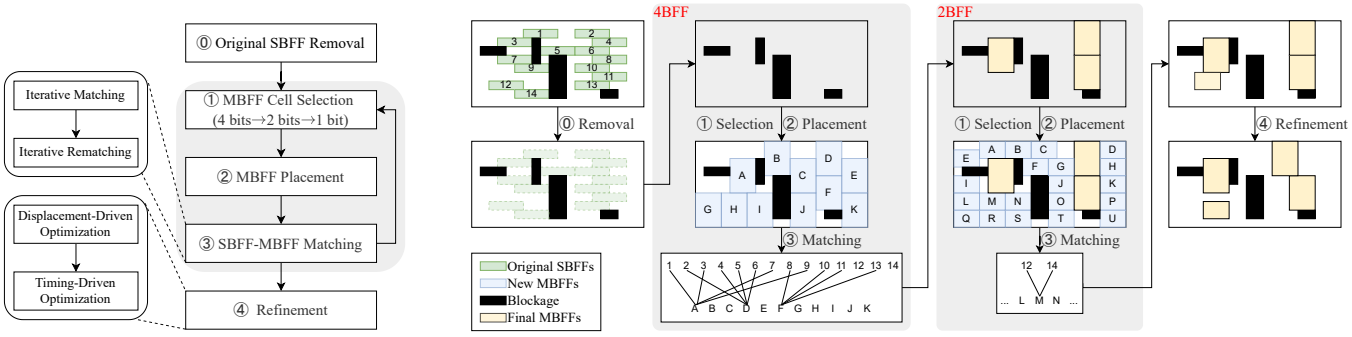
The flow of our method, depicted in Figure 3a, consists of five primary processes ⓪①②③④. Next, we illustrate these processes using the example shown in Figure 3b. The initial setup in Figure 3b consists of 14 SBFFs highlighted in green and four immovable combinational cells marked in black that act as blockage.

⓪ **Original SBFF Removal.** We start by removing all original SBFFs to create space for the MBFFs. With all SBFFs removed, each one must be matched to an MBFF that will be created subsequently. It is important to note that once an SBFF is removed, it cannot be restored to its original position, as that space may now be occupied by an MBFF.

**Iterations.** Next, we perform iterations of processes ①②③. Each iteration focuses on a distinct bitwidth of MBFFs in decreasing order (4, 2, and 1 in our setting). This sequence prioritizes MBFFs with larger bitwidths, which typically offer lower per-bit power consumption, thereby maximizing potential power reduction. Additionally, this order enhances the utilization of available space, as MBFFs with larger bitwidths occupy more area. It is important to note that the final iteration, which utilizes 1-bit "MBFFs"[2], may

[1] For brevity, we omit the detail regarding the clock-to-Q-pin delay of the starting FF in the critical path, which has negligible impact.

[2] Although a 1-bit FF is not technically an MBFF since it does not consist of multiple bits, we refer to all FFs other than the original SBFFs as "MBFFs" for ease of presentation.




**(a) Flow of our method** **(b) Example of the flow**

**Figure 3: Illustration of our method.**

offer limited benefits. However, it is essential to ensure that each SBFF can be matched to an MBFF. In extreme cases, the empty space created by the removal of SBFFs may be fragmented by immovable combinational cells, leaving only enough room for 1-bit FFs due to their smaller area. The use of 1-bit MBFFs is essential in such scenarios.

① **MBFF Cell Selection.** The cell library consists of multiple MBFF cells per bitwidth. To reduce the complexity, we select only one representative cell per bitwidth for MBFF placement and matching, leaving the extensive choices of same-bitwidth cells for the refinement stage for further optimization. Figure 3b shows two iterations, one with a 4-bit MBFF cell and the other with a 2-bit MBFF cell. This process will be detailed in Section 3.2.

② **MBFF Placement.** After an MBFF cell is selected for the current iteration, we try to place as many instances of the MBFF cell as possible to allow more SBFFs to be matched in this iteration. In Figure 3b, we successfully placed 11 instances (labeled A–K) during the 4BFF iteration, while 21 instances (labeled A–U) were placed during the 2BFF iteration. Our tiling-based placement approach is detailed in Section 3.3.

③ **SBFF-MBFF Matching.** This process focuses on matching the MBFF instances created in this iteration with unmatched original SBFFs. Our matching algorithm consists of *iterative matching* (Section 3.4) and *iterative rematching* (Section 3.5). It comprehensively considers (i) the number of SBFFs matched in this iteration, (ii) the increases in total displacement, and (iii) runtime efficiency. In Figure 3b, the 4-bit MBFF instances $A$, $D$, and $F$ from the 4BFF iteration are fully matched with SBFFs, while the 2-bit MBFF instance (M) from the 2BFF iteration is fully matched with 2 SBFFs.

④ **Refinement.** This process employs several strategies, including pin swapping, MBFF replacement, and MBFF relocation, to mitigate the TNS introduced by MBFF banking. These strategies are detailed in Section 3.6.

## 3.2 MBFF Cell Selection

For each bitwidth $b$, we select the $b$-bit MBFF cell that has the smallest weighted sum of power and area, where the weights for power and area are consistent with those used in the evaluation. This selection criterion is based on two key reasons. Firstly, power and area are the most critical metrics among all objectives (power, area, TNS, bin density violations) considered in this problem, contributing to over 90% of the overall score on average. Therefore, selecting the MBFF cell that has optimal power and area characteristics is more likely to yield superior banking solutions. Secondly, power and area can be estimated accurately and independently for each MBFF cell, making our selection criterion robust and reliable. In contrast, it is impossible to evaluate the timing aspect of an MBFF cell because timing calculations rely heavily on other cells on the critical paths. During MBFF cell selection, we do not know how many instances of the selected cell will be created and where they will be placed. We do not consider the metric of bin density violations for similar reasons.

**Algorithm 1** Tiling-Based Compact MBFF Placement

```
// Placing instances of MBFF cell F as many as possible on a X × Y grid
1: for x ← 1 to X do
2:     for y ← 1 to Y do
3:         if canPlace(F, x, y) then                ▷ Overlap checking
4:             Place(F, x, y)
5:         end if
6:     end for
7: end for
```

## 3.3 Tiling-Based MBFF Placement

Once an MBFF cell is selected, the goal is to place as many instances of this cell as possible on the grid. Maximizing the number of MBFF instances is crucial for enhancing quality, as it increases banking opportunities. However, the placement grid contains immovable combinational cells that must be avoided. These obstructions fragment the available space, posing challenges for efficient and effective MBFF placement.

To address this, we propose a tiling-based placement method, illustrated in Algorithm 1. Given a placement grid of dimensions $X \times Y$, with its lower left corner at $(1, 1)$, we evaluate each site $(x, y)$ on the grid sequentially from the lower left to the upper right (lines 1–2 in Algorithm 1). Each site is considered as a potential lower left corner for an MBFF instance. An instance is placed at $(x, y)$ whenever it does not result in any overlap (lines 3–4). This approach minimizes empty spaces between MBFF instances, ensuring a compact placement solution.

The efficiency of Algorithm 1 can be enhanced by eliminating unnecessary `canPlace` calls that are guaranteed to return false. When an MBFF instance is place at $(x, y)$, it occupies a rectangular area defined by the lower-left corner $(x, y)$ and the upper-right corner $(x + W - 1, y + H - 1)$, where $W$ and $H$ represent the width and height of the MBFF cell, respectively. Consequently, there is no need to invoke `canPlace` for positions within this occupied region.

## 3.4 Iterative Matching

After placing MBFF instances, we match original SBFFs with the newly placed MBFFs. The workflow is given in Algorithm 2, consisting of two key steps: *matching candidate generation* and *matching candidate selection* that are performed iteratively. The generation step produces a set $S$ of SBFF candidates for each MBFF. We first initialize these sets to empty (line 1 of Algorithm 2). We then check each pair of SBFF and MBFF and add it to the corresponding set if this pair satisfies certain conditions (lines 2–7). The conditions will be detailed in Section 3.4.1. After the generation step, the set $S$ for each MBFF may consist of more SBFFs than the bitwidth capacity of the MBFF, requiring further selection. For each MBFF, the selection step repeatedly picks an SBFF from its candidate set, which is closest to the MBFF, to form a matching, until the MBFF is fully matched (lines 9–14).

*3.4.1 Conditions for Candidate Generation.* In line 4 of Algorithm 2, we need to determine whether to include a pair of SBFF and MBFF as candidates. This decision is based on two conditions:

- First, the SBFF and MBFF must belong to the same clock net, which is essential for ensuring correctness. Newly created MBFFs from the previous MBFF placement phase do not have a clock-net attribute and can freely match with any SBFF. However, once an MBFF is matched with an SBFF, it inherits the clock-net attribute of the matched SBFF and must subsequently match only with SBFFs from the same clock net.
- Second, the distance between SBFF $s$ and MBFF $m$ must fall within a predefined search radius, defined as follows:

$$r = \frac{cost(s) - cost(m)}{\alpha \cdot \texttt{displacementDelay}}$$

where $cost(x)$ represents the per-bit weighted sum of area and power of cell $x$, expressed as

$$cost(x) = ((\beta \cdot power(x) + \gamma \cdot area(x))/\text{bitwidth}(x)$$

where $\beta$ and $\gamma$ are the weights for power and area. Essentially, this search radius ensures that the overall gain in terms of the weighted sum of TNS, area, and power remains non-negative. Specifically, $cost(s) - cost(m)$ reflects the gain in power and area achieved by banking SBFF $s$ to MBFF $m$. This gain is then divided by the timing weight $\alpha$ and `displacementDelay` to determine the maximum displacement whose negative impact on TNS can be offset by the power and area gain, ensuring an overall positive improvement.

## 3.5 Iterative Rematching

After the initial matching process, some MBFFs may remain only partially matched. For example, a 4-bit MBFF matched with three

**Algorithm 2** Matching Candidate Generation and Selection

```
1:  S(m) ← ∅ for every MBFF m
2:  for each unmatched SBFF s do                    ▷ Candidate Generation
3:      for each MBFF m do
4:          if check(s, m) = true then              ▷ Section 3.4.1
5:              S(m) ← S(m) ∪ {s}
6:          end if
7:      end for
8:  end for
9:  for each MBFF m do                              ▷ Candidate Selection
10:     while m is not fully matched and S(m) ≠ ∅ do
11:         match m to s ∈ S(m) that minimizes distance(m, s)
12:         Remove s from S(m)
13:     end while
14: end for
```

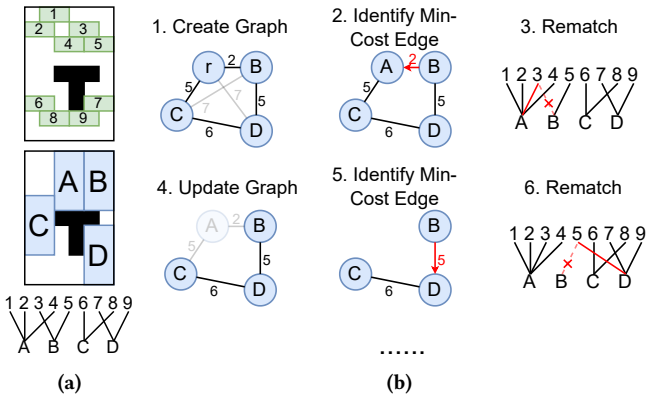



**Figure 4: Rematching for partially matched MBFFs. (a) Original matching results, where all 4-bit MBFFs (labeled A–D) are only partially matched, meaning each is connected to fewer than 4 SBFFs. (b) Iterative rematching process: this involves identifying the closest pair of MBFFs and transferring an SBFF from one MBFF to the other iteratively.**

SBFFs is partially matched, as it has one unmatched bit. Partially-matched MBFFs cannot fully utilize resources, leading to suboptimal performance. The *rematching* process aims to adjust matchings among partially-matched MBFFs to achieve either fully-matched MBFFs or fully unmatched MBFFs that can be removed. To accomplish this, we iteratively identify existing matchings between an SBFF and a partially-matched MBFF, remove these matchings, and rematch the SBFF to another partially-matched MBFF. We illustrate this process using the example in Figure 4a, which involves 9 SBFFs (labeled 1–9) matched to 4 MBFFs (labeled A–D). The MBFF-SBFF matching is depicted at the bottom of Figure 4a.

**Graph Construction.** We represent the partially-matched MBFFs as a graph, where each node corresponds to an MBFF. An edge exists between two MBFFs if they are within a precomputed Manhattan distance. This distance threshold is calculated similarly to the search radius outlined in Section 3.4.1. The distance between two MBFFs also serves as the edge cost, indicating the potential displacement incurred when transferring an SBFF along that edge. Figure 4b illustrates the constructed graph in step 1, where edges with a cost greater than 6 are excluded due to their high expense.

**Iterative Rematching.** We start by selecting the minimum-cost edge from the graph, such as the edge between $A$ and $B$ in step 2 of Figure 4b. The direction of the selected edge is determined by pointing from the MBFF with fewer matched SBFFs to the one with more; in this case, the direction is $B \rightarrow A$ since $A$ is matched to 3 SBFFs while $B$ is matched to 2. We then remove the matching of $B$ and rematch the corresponding SBFF to $A$, as shown in step 3 of Figure 4b. After this rematching, we update the graph by removing MBFFs that have become either fully matched or fully unmatched, followed by another iteration of rematching. This process continues until no more edges can be selected.

**Removal of Partially-Matched MBFFs.** After the iterative rematching process, any remaining partially-matched MBFFs will be eliminated, leaving the associated SBFFs unmatched. These unmatched SBFFs will be addressed in subsequent banking iterations. It is important to note that in the final banking iteration, the MBFF cell is 1 bit, which means there will be no partially-matched MBFFs. Consequently, this removal phase will not be activated, ensuring a correct final solution.

## 3.6 Refinement

The banking stage banks all original SBFFs to MBFFs to optimize power and area. However, this process can negatively impact timing due to the relocation of original SBFF pins. The *refinement* stage aims to mitigate this adverse timing effect through a series of targeted adjustments. We begin with displacement-driven refinement for quick initial optimization, followed by timing-driven refinement for further improvement. Although displacement is not the primary focus, reducing it can minimize disruption to the original optimized placement, thereby improving TNS. Given that refinement is primarily aimed at enhancing timing, it may not significantly benefit designs that are not timing-critical. For instance, in some benchmarks of the ICCAD Contest [14], TNS accounts for less than 1% of the total quality score. Consequently, we only perform refinement on designs that are timing-critical.

*3.6.1 Displacement-Driven Optimization.* The displacement of an SBFF is defined as the Manhattan distance between the location of its pin and that of the banked MBFF pin. We use three techniques to reduce displacement: *MBFF relocation*, *pin swapping*, and *MBFF replacement*. MBFF relocation involves finding a new available location for an MBFF, while pin swapping involves exchanging the locations of two pins, which may belong to the same MBFF or different ones. MBFF replacement involves substituting an MBFF with another MBFF cell of the same bitwidth from the cell library. The new MBFF cell features a different set of pin locations within the MBFF cell, potentially resulting in reduced displacement.

*3.6.2 Timing-Driven Optimization.* To facilitate timing-driven optimization, we have implemented a timing calculator that provides crucial feedback on timing results. We employ MBFF relocation and MBFF debanking to reduce TNS. While MBFF relocation follows a similar strategy to displacement-driven optimization, the focus shifts from minimizing displacement to optimizing TNS. MBFF debanking involves decomposing an MBFF into multiple SBFFs, allowing the pins to be placed individually and potentially at a distance for improved TNS.

**Table 1: Testcase Details**

| Case | #Cells | #Nets | #FFs | $\alpha$ | $\beta$ | $\gamma$ | $\lambda$ |
|---|---|---|---|---|---|---|---|
| 1 | 108685 | 104235 | 19879 | | 2E+03 | 2E-03 | |
| 2 | 153457 | 153934 | 21164 | | 4E+02 | 8E-07 | |
| 3 | 101221 | 104761 | 13016 | | 1E+04 | 2E-03 | |
| 4 | 108685 | 104235 | 19879 | 1E+01 | 2E+05 | 2E-06 | 1E+04 |
| 5 | 153457 | 153934 | 21164 | | 4E+04 | 8E-07 | |
| 6 | 153457 | 153934 | 21164 | | 4E+02 | 8E-05 | |
| 7 | 101221 | 104761 | 13016 | | 1E+04 | 2E-03 | |

# 4 EXPERIMENTAL RESULTS

We implemented our method in C++ and conducted experiments on an Ubuntu 22.04 server equipped with INTEL(R) XEON(R) PLATINUM 8575C CPUs. For comparison, we evaluated our approach against two leading works: the first-place team from the ICCAD Contest [14] and TIMBER [1]. To ensure a fair comparison, we obtained the executables of these works and executed them in the same environment. The quality scores were reported using the official evaluator of the Contest.

## 4.1 Benchmarks

Table 1 presents the specifics of seven test cases used in our experiments, consistent with the settings of the Contest [14]. The weights $\alpha$, $\beta$, $\gamma$, and $\lambda$ correspond to TNS, power, area, and bin density violations, respectively. The quality score of a solution is computed using the formula:

$$\sum_{\forall i \in \mathrm{FF}} (\alpha \cdot \mathrm{TNS}(i) + \beta \cdot \mathrm{Power}(i) + \gamma \cdot \mathrm{Area}(i)) + \lambda \cdot BDV$$

where $BDV$ represents the number of bins exceeding the density requirement. The methods for calculating TNS, power, area, and BDV are detailed in Section 2.

## 4.2 Quality and Runtime Comparisons

Table 2 presents a comparison of the quality scores achieved by the first-place team, TIMBER [1], and our approach. It is important to note that our timing-driven refinement is not enabled by default, as most cases are not timing-critical. Timing-driven optimization will be evaluated in Section 4.4.2. Our method achieves the minimum geometric mean results in terms of both TNS and total score, demonstrating the effectiveness of our overall framework.

The runtime comparisons are detailed in Table 3. While TIMBER [1] is the fastest, this speed comes at the expense of quality, as shown in Table 2. Notably, our method exhibits slightly faster performance than the first-place team.

## 4.3 Robustness Comparisons

To evaluate robustness, we conducted additional experiments by varying the weights. Case 6 was selected for this analysis because all methods yielded similar scores (within 1%) under its default weights, providing a fair baseline. We separately increased the TNS ($\alpha$) and power weight ($\beta$) by factors of 2, 4 and 8. Experiments for the area weight ($\gamma$) are not included, as the area score is the dominant factor in case 6, and further increasing $\gamma$ made no difference. The effects of adjusting $\alpha$ and $\beta$ are illustrated in Figure 5. TIMBER [1] and

**Table 2: Comparison of quality scores among the first-place team (1st) of the ICCAD Contest [14], TIMBER [1], and this work. BDV stands for Bin Density Violation, and lower scores indicate better performance.**

| Case | TNS | | | Power | | | Area | | | BDV ($\times 10^4$) | | | Total Score | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1st | [1] | Ours | 1st | [1] | Ours | 1st | [1] | Ours | 1st | [1] | Ours | 1st | [1] | Ours |
| 1 | 2566980 | 6337210 | 2359370 | 372402 | 393148 | 402966 | 735346458 | 741213396 | 736605954 | 1 | 0 | 0 | 738295840 | 747943754 | 739368290 |
| 2 | 63384 | 830642 | 94697 | 119900 | 72710 | 147770 | 557408 | 556252 | 559190 | 0 | 0 | 0 | 740692 | 1459604 | 801657 |
| 3 | 3010470 | 237527 | 1588970 | 1403530 | 16368500 | 1408270 | 724499370 | 777970116 | 724525074 | 1 | 0 | 3 | 728923370 | 779844557 | 727552314 |
| 4 | 906089 | 485074 | 477488 | 30044000 | 37822400 | 30914999 | 750546 | 783391 | 770640 | 14 | 0 | 0 | 31840635 | 39090865 | 32163127 |
| 5 | 6109510 | 8379930 | 4185020 | 7880000 | 7271000 | 7136000 | 556708 | 556252 | 555994 | 4 | 0 | 1 | 14586218 | 16207182 | 11887014 |
| 6 | 566059 | 830642 | 344445 | 81950 | 72710 | 79820 | 55620472 | 55625287 | 55623974 | 4 | 0 | 0 | 56308481 | 56528639 | 56048239 |
| 7 | 2933320 | 237443 | 1663180 | 1405380 | 16338500 | 1408270 | 724488660 | 778072932 | 724525074 | 1 | 0 | 3 | 728837360 | 779944257 | 727626524 |
| Geomean | 1239789 | 1000344 | 879302 | 1079543 | 2051370 | 1110256 | 24297845 | 24968281 | 24402860 | - | - | - | 69198547 | 81449713 | 68001181 |
| Ratio | 1.41 | 1.14 | 1.00 | 0.97 | 1.85 | 1.00 | 1.00 | 1.02 | 1.00 | - | - | - | 1.02 | 1.20 | **1.00** |

**Table 3: Comparison of Running Time in Seconds**

| Method | 1 | 2 | 3 | 4 | 5 | 6 | 7 | Avg |
|---|---|---|---|---|---|---|---|---|
| 1st Place | 3.83 | 27.05 | 2.67 | 2.52 | 27.18 | 4.31 | 2.76 | 10.05 |
| TIMBER[1] | 1.71 | 2.85 | 2.49 | 2.41 | 2.84 | 2.85 | 2.46 | 2.52 |
| Ours | 3.86 | 16.07 | 11.63 | 2.12 | 16.54 | 5.87 | 11.74 | 9.69 |

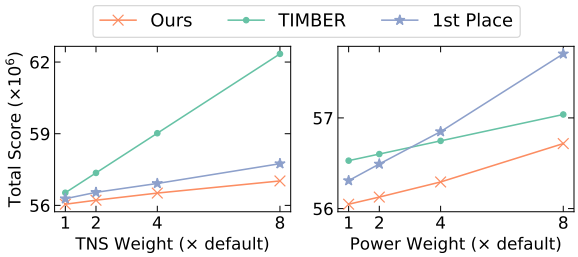



**Figure 5: Case-6 results under different weighting schemes.**

the first-place method show a rapid increase in score with higher $\alpha$ (left) and $\beta$ (right), respectively. Our method shows relatively slow score growth as both $\alpha$ and $\beta$ increase, demonstrating its robustness across different weighting schemes.

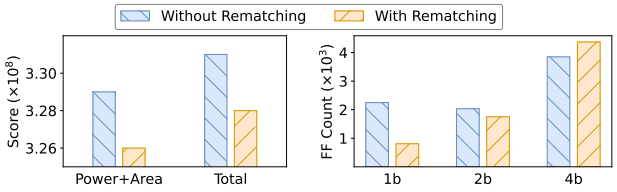



**Figure 6: Average results with and without rematching.**

## 4.4 Effectiveness of Rematching & Refinement

*4.4.1 Rematching.* We assessed the effectiveness of the rematching process described in Section 3.5 by conducting additional experiments that omitted rematching (i.e., partially matched MBFFs were directly removed). Figure 6 presents the average scores and the FF bitwidth counts across all cases. The results illustrate that rematching enables the utilization of large-bitwidth MBFFs, leading to improved overall scores by reducing both power and area metrics.

**Table 4: Results of Case 5 (Timing Critical) with and without Displacement-Driven (D) and Timing-Driven (T) Refinement.**

| Method | Refinement | | Score (Ratio) | | Time |
|---|---|---|---|---|---|
| | D | T | TNS | Total | |
| 1st Place | - | - | 6109510 (2.13) | 14586218 (1.35) | 27.18 |
| TIMBER[1] | - | - | 8379930 (2.92) | 16207182 (1.50) | 2.84 |
| Ours | × | × | 6314100 (2.20) | 14016094 (1.30) | 12.28 |
| | ✓ | × | 4185020 (1.46) | 11887014 (1.10) | 16.54 |
| | ✓ | ✓ | 2866930 (1.00) | 10790489 (1.00) | 71.99 |

*4.4.2 Refinement.* We select case 5, the most timing-critical case, for refinement analysis. Table 4 shows its experimental results with different refinement strategies. Even without refinement, our method already outperforms the other two approaches. Enabling displacement-driven optimization substantially reduces the TNS and total score, with minimal impact on runtime. Timing-driven refinement further enhances the results at the cost of longer runtime.

## 5 CONCLUSION

This paper introduces a novel algorithm for MBFF banking that first places MBFFs compactly on the grid, followed by an iterative matching of original SBFFs to these newly placed MBFFs. Unlike conventional clustering-based methods, our approach prioritizes compact placement to accommodate a greater number of MBFFs, thereby enhancing banking opportunities and reducing power consumption. Experimental results demonstrate improvements in overall quality ranging from 2% to 20% compared to previous methods.

A limitation of our approach is its runtime efficiency. Inspired by recent GPU-accelerated physical design algorithms [6–10], leveraging GPUs to accelerate our method is a promising solution.

## ACKNOWLEDGMENTS

This work was partially supported by Guangdong Provincial Project 2024QN11X113 and GDIC.

# REFERENCES


[1] Aditya Das Sarma, Shui Jiang, Wan Luan Lee, Tsung-Yi Ho, and Tsung-Wei Huang. 2026. TIMBER: A Fast Algorithm for Timing and Power Optimization using Multi-bit Flip-flops. In *2026 31st Asia and South Pacific Design Automation Conference (ASP-DAC)*.

[2] S.-Y. Fang, Y.-Y. Liu, C.-K. Cheng, and T.-M. Tseng. 2025. Overview of 2024 CAD Contest at ICCAD. In *Proc. 43rd IEEE/ACM Int. Conf. Comput.-Aided Des.* (Newark Liberty Int. Airport Marriott, New York, NY, USA) *(ICCAD '24)*. ACM Press, New York, NY, USA, Article 195, 3 pages. https://doi.org/10.1145/3676536.3689908

[3] Y.-T. Hsieh, T.-W. Lin, Y.-C. Chang, H.-Y. Wu, and I. H.-R. Jiang. 2025. Graceful Register Clustering and Rebanking for Power and Timing Balancing. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* 44, 3 (2025), 1070–1083. https://doi.org/10.1109/TCAD.2024.3448233

[4] A. B. Kahng, S. Kundu, and S. Thumathy. 2024. Scalable Flip-Flop Clustering Using Divide and Conquer for Capacitated K-Means. In *Proc. Great Lakes Symp. VLSI 2024* (Clearwater, FL, USA) *(GLSVLSI '24)*. ACM Press, New York, NY, USA, 177–184. https://doi.org/10.1145/3649476.3658744

[5] H.-Y. Kao, C.-H. Hsu, and S.-H. Huang. 2019. Two-Stage Multi-Bit Flip-Flop Clustering with Useful Skew for Low Power. In *Proc. 2019 2nd Int. Conf. Commun. Eng. Technol.* IEEE Computer Society, Washington, DC, USA, 178–182. https://doi.org/10.1109/ICCET.2019.8726883

[6] Shiju Lin, Jinwei Liu, Evangeline F. Y. Young, and Martin D. F. Wong. 2023. GAMER: GPU-Accelerated Maze Routing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42, 2 (2023), 583–593. https://doi.org/10.1109/TCAD.2022.3184281

[7] Shiju Lin and Martin D. F. Wong. 2022. Superfast Full-Scale GPU-Accelerated Global Routing. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design* (San Diego, California) *(ICCAD '22)*. Association for Computing Machinery, New York, NY, USA, Article 51, 8 pages. https://doi.org/10.1145/3508352.3549474

[8] Shiju Lin, Liang Xiao, Jinwei Liu, and Evangeline F. Y. Young. 2025. InstantGR: Scalable GPU Parallelization for Global Routing. In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design* (Newark Liberty International Airport Marriott, New York, NY, USA) *(ICCAD '24)*. Association for Computing Machinery, New York, NY, USA, Article 148, 8 pages. https://doi.org/10.1145/3676536.3676787

[9] Yibo Lin, Shounak Dhar, Wuxi Li, Haoxing Ren, Brucek Khailany, and David Z. Pan. 2019. DREAMPlace: Deep Learning Toolkit-Enabled GPU Acceleration for Modern VLSI Placement. In *Proceedings of the 56th Annual Design Automation Conference 2019* (Las Vegas, NV, USA) *(DAC '19)*. Association for Computing Machinery, New York, NY, USA, Article 117, 6 pages. https://doi.org/10.1145/3316781.3317803

[10] Lixin Liu, Bangqi Fu, Shiju Lin, Jinwei Liu, Evangeline F. Y. Young, and Martin D. F. Wong. 2024. Xplace: An Extremely Fast and Extensible Placement Framework. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 43, 6 (2024), 1872–1885. https://doi.org/10.1109/TCAD.2023.3346291

[11] M.-Y. Liu, Y.-C. Lai, W.-K. Mak, and T.-C. Wang. 2022. Generation of Mixed-Driving Multi-Bit Flip-Flops for Power Optimization. In *Proc. 2022 IEEE/ACM Int. Conf. Comput.-Aided Des.* ACM Press, New York, NY, USA, 1–9.

[12] P. Vanna-Iampikul, Y.-C. Lu, D. E. Shim, and S. K. Lim. 2023. GNN-Based Multi-Bit Flip-Flop Clustering and Post-Clustering Design Optimization for Energy-Efficient 3D ICs. *ACM Trans. Des. Autom. Electron. Syst.* 28, 5, Article 76 (Sept. 2023), 26 pages. https://doi.org/10.1145/3588570

[13] G. Wu, Y. Xu, D. Wu, M. Ragupathy, Y.-Y. Mo, and C. Chu. 2016. Flip-Flop Clustering by Weighted K-Means Algorithm. In *Proc. 2016 53rd ACM/EDAC/IEEE Des. Autom. Conf.* ACM Press, New York, NY, USA, 1–6. https://doi.org/10.1145/2897937.2898025

[14] S.-W. Yang, J.-W. Hsu, T.-W. Lee, T.-H. Chen, and C.-F. C. Shen. 2025. 2024 ICCAD CAD Contest Problem B: Power and Timing Optimization Using Multibit Flip-Flop. In *Proc. 43rd IEEE/ACM Int. Conf. Comput.-Aided Des.* (Newark Liberty Int. Airport Marriott, New York, NY, USA) *(ICCAD '24)*. ACM Press, New York, NY, USA, Article 197, 6 pages. https://doi.org/10.1145/3676536.3689911